METHOD FOR FAST COMPILATION OF PREVERIFIED JAVA™ BYTECODE TO HIGH QUALITY NATIVE MACHINE CODE

By Inventor

Beat Heeb

CROSS REFERENCE TO RELATED APPLICATIONS

[0001.]This Application claims the benefit of U.S. Provisional Application No. 60/294,913 filed 5/31/2001, the disclosure of which is incorporated herein by reference.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH

[0002.]Not applicable.

BACKGROUND OF INVENTION

FIELD OF INVENTION

[0003.]The present invention is related to the compilation of platform neutral bytecode computer instructions, such as JAVA, into high quality machine code. More specifically, the present invention discloses a new method of creating high quality machine code from platform neutral bytecode in a single sequential pass in which information from preceding instruction translations is used to mimic an optimizing complier without the extensive memory and time requirements.

BACKGROUND OF INVENTION

[0004.]The benefit of architecture neutral language such as JAVA is the ability to execute such language on a wide range of systems once a suitable implementation technique, such as a JAVA Virtual Machine, is present. The key feature of the JAVA language is the creation and use of platform neutral bytecode instructions, which create the ability to run JAVA programs, such as applets, applications or servelets, on a broad range of diverse platforms. Typically, a JAVA program is compiled through the use of a

JAVA Virtual Machine (JVM) which is merely an abstract computing machine used to compile the JAVA program (or source code) into platform neutral JAVA bytecode instructions, which are then placed into class files. The JAVA bytecode instructions in turn, serve as JVM instructions wherever the JVM is located. As bytecode instructions, the JAVA program may now be transferred to and executed by any system with a compatible JAVA platform. In addition, any other language which may be expressed in bytecode instructions, may be used with the JVM.

[0005.]Broadly speaking, computer instructions often are incompatible with other computer platforms. Attempts to improve compatibility include "high level" language software which is not executable without compilation into a machine specific code. As taught by U.S. Patent No. 5,590,331, issued December 31, 1996 to Lewis et al., several methods of compilation exist for this purpose. For instance, a pre-execution compilation approach may be used to convert "high level" language into machine specific code prior to execution. On the other hand, a runtime compilation approach may be used to convert instructions and immediately send the machine specific code to the processor for execution. A JAVA program requires a compilation step to create bytecode instructions, which are placed into class files. A class file contains streams of 8-bit bytes either alone or combined into larger values, which contain information about interfaces, fields or methods, the constant pool and the magic constant. Placed into class files, bytecode is an intermediate code, which is independent of the platform on which it is later executed. A single line of bytecode contains a one-byte opcode and either zero or additional bytes of operand information. Bytecode instructions may be used to control stacks, the VM register arrays or transfers. A JAVA interpreter is then used to execute the compiled bytecode instructions on the platform.

[0006.]The compilation step is accomplished with multiple passes through the bytecode instructions, where during each pass, a loop process is employed in which a method loops repeatedly through all the bytecode instructions. A single bytecode instruction is analyzed during each single loop through the program and after each loop, the next loop through the bytecode instructions analyzes the next single bytecode instruction. This is repeated until the last bytecode instruction is reached and the loop is ended.

[0007.]During the first compilation pass, a method loops repeatedly through all the bytecode instructions and a single bytecode instruction is analyzed during each single loop through the program. If it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended. If the bytecode instruction being analyzed is not the last bytecode instruction, the method determines stack status from the bytecode instruction and stores this in stack status storage, which is updated for each bytecode instruction. This is repeated until the last bytecode instruction is reached and the loop is ended.

[0008.]During the second compilation pass, a method loops repeatedly through all the bytecode instructions once again and a single bytecode instruction is analyzed during each single loop through the program. If it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended. If the bytecode instruction being analyzed is not the last bytecode instruction, the stack status storage and bytecode instruction are used to translate the bytecode instruction into machine code. This is repeated until the last bytecode instruction is translated and the loop is ended.

[0009.]A JAVA program however, also requires a verification step to ensure malicious or corrupting code is not present. As with most programming languages, security concerns are addressed through verification of the source code. JAVA applications ensure security through a bytecode verification process which ensures the JAVA code is valid, does not overflow or underflow stacks, and does not improperly use registers or illegally convert data types. The verification process traditionally consists of two parts achieved in four passes. First, verification performs internal checks during the first three passes, which are concerned solely with the bytecode instructions. The first pass checks to ensure the proper format is present, such as bytecode length. The second pass checks subclasses, superclasses and the constant pool for proper format. The third pass actually verifies the bytecode instructions. The fourth pass performs runtime checks, which confirm the compatibility of the bytecode instructions.

[00010.]As stated, verification is a security process, which is accomplished through several passes. The third pass in which actual verification occurs, employs a loop process similar to the compilation step in which a method loops repeatedly through

all the bytecode instructions and a single bytecode instruction is analyzed during each single loop through the program. After each loop, the next loop through the bytecode instructions analyzes the next single bytecode instruction which is repeated until the last bytecode instruction is reached and the loop is ended.

[0011.]During the verification pass, the method loops repeatedly through all the bytecode instructions and a single bytecode instruction is analyzed during each single loop through the program. If it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended. If the bytecode instruction is not the last bytecode instruction, the position of the bytecode instruction being analyzed is determined. If the bytecode instruction is at the beginning of a piece of code that is executed contiguously (a basic block), the global stack status is read from bytecode auxiliary data and stored. After storage, it is verified that the stored global stack status is compliant with the bytecode instruction. If however, the location of the bytecode instruction being analyzed is not at the beginning of a basic block, the global stack status is not read but is verified to ensure the global stack status is compliant with the bytecode instruction. After verifying that the global stack status is compliant with the bytecode instruction, the global stack status is changed according to the bytecode instruction. This procedure is repeated during each loop until the last bytecode instruction is analyzed and the loop ended.

[0012.]It may be noted that the pass through the bytecode instructions that is required for verification closely resembles the first compilation pass. Duplicate passes during execution can only contribute to the poor speed of JAVA programs, which in some cases may be up to 20 times slower than other programming languages such as C. The poor speed of JAVA programming is primarily the result of verification. In the past, attempts to improve speed have included compilation during idle times and pre-verification. In U.S. Patent No. 5,970,249 issued October 19,1999 to Holzle et al., a method is taught in which program compilation is completed during identified computer idle times. And in U.S. Patent No. 5,999,731 issued December 7, 1999 to Yellin et al. the program is pre-verified, allowing program execution without certain determinations such as stack overflow or underflow checks or data type checks. Both are attempts to improve execution speed by manipulation of the compilation and verification steps. In order to

further improve speed, a method and apparatus is needed that can combine these separate, yet similar steps, the verification pass, and the first and second compilation pass, into a step which accomplishes the multiple tasks in substantially less time.

BRIEF DESCRIPTION OF DRAWINGS

[0013.]These and other objects, features and characteristics of the present invention will become more apparent to those skilled in the art from a study of the following detailed description in conjunction with the appended claims and drawings, all of which form a part of this specification. In the drawings:

FIG. 1A (prior art) illustrates a flowchart of traditional bytecode instruction first pass compilation;

FIG. 1B (prior art) illustrates a flowchart of traditional bytecode instruction second pass compilation;

FIG. 2 illustrates a flowchart of the embodiment of the new method; and

FIG. 3 illustrates the data structures required by the new method.

DETAILED DESCRIPTION OF PRESENTLY PREFERRED EXEMPLARY EMBODIMENTS

[0014.]It is the object of the present invention to create a method and apparatus which may be used to compile platform independent bytecode into high quality native machine code without the extensive memory and execution time requirements of an optimizing compiler. The present invention produces the fast and simple translation associated with simple translators, but with a resulting translation equivalent to the high quality output associated with the more complex optimizing compiler. The new method consists of a program instruction set which executes fewer passes through a bytecode instruction listing where complete compilation into high quality native machine code is achieved in less time and requiring less memory.

[0015.]The new method translates bytecode instruction listings into native machine code in a single sequential pass. Each bytecode instruction is handled sequentially using information remembered from the translation of preceding bytecode instructions. However, where no direct control flow from the previous instruction prevents information from preceding instructions to be used, information extracted from the stack maps contained in all preverified class files may be used instead.

[0016.]In prior art Figures 1A and 1B, an illustrative flow diagram of a simple bytecode translator compilation method is shown. In prior art Figure 1A, a traditional compilation method is shown as flow diagram 100 which loops through the bytecode instructions, analyzing an individual bytecode instruction during each loop as stated in step 102. After each bytecode instruction is analyzed, the method determines the stack status from the bytecode instruction being analyzed and stores the stack status in stack status storage as stated in step 104. When the last bytecode instruction is analyzed as stated step 102, the loop is ended at step108 and partial compilation is completed.

[0017.]In prior art Figure 1B, remaining compilation occurs in flow diagram 150 which shows further loops through the bytecode instructions analyzing an individual bytecode instruction during each loop as stated in step 152. The stack status storage and bytecode instruction are then used to translate the bytecode instruction into machine code

as stated in step 154. When the last bytecode instruction is translated as stated in step 152, the loop is ended at step 158 and compilation is completed.

[0018.]In Figure 2, an illustrative flow diagram of the new method is shown in flow diagram 200. In step 202, a class file placed on the development or target system is selected and a first method within the first class file is selected in step 204. At this point, storage and initialization occurs in step 206. In step 208 a first bytecode instruction is selected and evaluated to determine if there is a stack map stored for the actual bytecode instruction and, if there is a stored stack map, step 210 determines if there is direct control flow from the previous instruction. If there is no direct control flow present, step 214 results in reading the stack layout from the stack map in bytecode and setting mappings to 'stack'. If direct control flow is present, step 212 produces a code to store all stacks and set their stack mappings to 'stack'. In step 216, the native code address for the actual instruction is then set.

[0019.]In steps 218 and 222, the instruction is evaluated to determine if the actual instruction is 'load constant' or load local'. If the instruction is load constant, step 220 creates a new constant stack mapping. If the instruction is load local, step 224 creates a new local stack mapping. If the instruction is a stack manipulating instruction as determined in step 226, stack mappings are duplicated or reordered in step 228 according to the instruction. If the actual instruction is a jump or switch instruction as determined in step 230, a code is produced for the actual instruction using stack mapping information and a code is produced to store all stack values not used in step 232.

[0020.]In step 234, if the actual instruction is any other instruction , a code is produced for the actual instruction using stack mapping information to locate the arguments in step 236. The mappings for the arguments are removed and new mappings are created if the instruction results in a new value.

[0021.]Once the instruction has been analyzed, the following bytecode instruction is selected for translation. If there are no remaining instructions, the next method is selected in step 204 and if there are no remaining methods, the next class is selected in step 202. If there are no remaining classes, the evaluation returns in step 238.

[0022.]Several data structures are required to remember the information of the preceding instruction translation in the new method. For each possible value currently

available on the bytecode stack, a field is required showing the actual mapping to storage locations in the target machine architecture, as well as a field containing additional information on the first field. A field showing actual mapping to storage locations is required for constants, locals, temporaries and stacks. The second field contains additional information such as constant values, slot and register numbers. For each bytecode address which is the target of a jump or switch instruction, an additional field is required to store the corresponding native address code.

[0023.]Referring now to Table 1, the new fast compilation method places each class file in the development or target system, at which point each method in the class containing bytecode instructions is analyzed. Storage and data structures for actual mappings and native code addresses are created and the stack mappings are initialized to empty and addresses are initialized to unknown. Each bytecode instruction, from first to last is then evaluated and translated into high quality machine code.

[0024.]Each bytecode instruction is evaluated sequentially from first to last, and starting with the first, the new method determines if there is a stack map stored for the actual bytecode instruction. If a stack map is stored for the actual instruction, the new method then determines if there is direct control flow from the previous instruction (for each bytecode instruction after the first). If direct control flow exists, a code is produced to store all stacks and set their stack mapping to 'stack'. If no direct control flow exists, the stack layout in bytecode is read from the stack map and mappings are set to 'stack'. Once the code is produced or stack layout is read, the native code address is set for the actual instruction.

[0025.]The sequential bytecode instructions are then evaluated to determine if the actual instruction is 'load constant', load local', a stack manipulating instruction, a jump, switch or any other instruction. If the actual instruction is load constant, new constant stack mapping is created. If however, the actual instruction is load local, new local stack mapping is created.

[0026.]If the actual instruction is a stack manipulating instruction such as pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2, or swap, stack mappings are duplicated or reordered according to the actual instruction. If the actual instruction is a

jump or switch instruction, a code is produced for the actual instruction using stack mapping information to locate the arguments and native code addresses to get the actual destination address and the mappings for the arguments are removed.. A code is also produced to store all stack values not used by this instruction and their stack mapping is set to 'stack'.

[0027.]If the actual instruction is any other instruction, a code is produced for the actual instruction using stack mapping information to locate the arguments. The mappings for the arguments are removed and a new mapping is created if the instruction results in a new value. The process is repeated for each method within each class file, and thereafter repeated for each class file.

[0028.]Prior art methods such as simple translators and optimizing compilers fail to produce the results associated with the new method. Through the use of a sequential pass, the simplicity and speed of simple translators is achieved. However the use of preceding translation information to mimic optimizing compilers when possible, creates the high quality machine code translation of the optimizing compiler. Therefore the present invention produces a high quality translation with greater simplicity and speed than previously known.